ICS 2011
12th INFORMS Computing Society Conference

© 2011 INFORMS | ISBN 978-0-9843378-1-1
DOI 10.1287/ics.2011.0010

Lookahead Branching for Mixed Integer Programming

Wasu Glankwamdee
IBM, Singapore, Singapore 486048, wasu@sg.ibm.com

Jeff Linderoth
Department of Industrial and Systems Engineering, University of Wisconsin-Madison, Madison, Wisconsin 53706, linderoth@wisc.edu

Abstract We consider the effectiveness of a *lookahead* branching method for the selection of branching variable in branch-and-bound method for mixed integer programming. Specifically, we ask the following question: by taking into account the impact of the current branching decision on the bounds of the child nodes *two* levels deeper than the current node, can better branching decisions be made? We describe methods for obtaining and combining bound information from two levels deeper in the branch-and-bound tree, demonstrate how to exploit auxiliary implication information obtain in the process, and provide extensive computational experience showing the effectiveness of the new method. Our results show that the new search method can often significantly reduce the number of nodes in the search tree, but the computational overhead of obtaining information two levels deeper typically outweighs the benefits.

Keywords mixed integer linear programming; branch and bound; strong branching

1. Introduction

A mixed integer program (MIP) is the problem of finding

$$z_{\text{MIP}} = \max\{c^T x + h^T y\colon Ax + Gy \le b,\ x \in \mathbb{Z}^{|I|},\ y \in \mathbb{R}^{|C|}\}, \tag{MIP}$$

where I is the set of integer-valued decision variables, and C is the set of continuous decision variables. The most common algorithm for solving MIP, due to Land and Doig [23], is a branch-and-bound method that uses the linear programming relaxation of MIP to provide an upper bound on the optimal solution value (z_{MIP}). Based on the solution of the relaxation, the feasible region is partitioned into two or more subproblems. The partitioning processes is repeated, resulting in a tree of relaxations (typically called a *branch-and-bound tree*) that must be evaluated in order to solve MIP. See Nemhauser and Wolsey [29] or Wolsey [34] for a more complete description of the branch-and-bound method for MIP.

A key decision impacting the effectiveness of the branch-and-bound method is *how* to partition the feasible region. Typically, the region is divided by *branching on a variable*. Branching on a variable is performed by identifying a decision variable x_j whose solution value in the relaxation ($\hat{x}_j$) is not integer-valued. The constraint $x_j \le \lfloor \hat{x}_j \rfloor$ is enforced in one subproblem, and the constraint $x_j \ge \lceil \hat{x}_j \rceil$ is enforced in the other subproblem. In a given solution $(\hat{x}, \hat{y})$ to the LP relaxation of MIP, there may be many decision variables for which $\hat{x}_j$ is fractional. A branching method prescribes on *which* of the fractional variables to base the branching dichotomy.

The effectiveness of the branch-and-bound method strongly depends on how quickly the upper bound on z_{MIP}, obtained from the solution to a relaxation, decreases. Therefore, we would like to branch on a variable that will reduce this upper bound as quickly as possible. In fact, a long line of integer programming research in the 1970's was focused on developing branching methods that estimated which variables would be most likely to lead to a large decrease in the upper bound of the relaxation after branching (Bénichou et al. [6], Breu and Burdet [8], Forrest et al. [17], Gauthier and Ribière [18], Geoffrion and Marsten [19], Mitra [28]).

In the 1990's, in connection with their work on solving large-scale instances of traveling salesperson problems, Applegate et al. proposed the concept of *strong branching* (Applegate et al. [4]). In strong branching, the selection of a branching variable is made by first selecting a set $\mathcal{C}$ of candidates. Each variable in the candidate set is tentatively (and partially) branched on by performing a fixed, limited number of dual simplex pivots on the resulting child nodes. The intuition behind strong branching is that if the subproblem bounds change significantly in a limited number of simplex pivots, then the bounds will also change significantly (relative to other choices) should the child node relaxations be fully resolved. Strong branching has been shown to be an effective branching rule for many MIP instances and has been incorporated into many commercial solvers, e.g., CPLEX (CPLEX Optimization [10]), FICO-Xpress (Dash Optimization [11]). In *full strong branching*, the set $\mathcal{C}$ is chosen to be the set of all fractional variables in the solution of the relaxation, and there is no upper limit placed on the number of dual simplex pivots performed. Full strong branching is a computationally expensive method, so typically $\mathcal{C}$ is chosen to be a subset of the fractional variables in the relaxation solution, and the number of simplex pivots performed is small.

The fact that strong branching can be a powerful, but computational costly, technique has led some researchers to consider weaker forms of strong branching that only perform the necessary computations at certain nodes. For example, Linderoth and Savelsbergh [25] suggest performing the strong branching computations for variables that have yet to be branched upon. The commercial package LINDO performs strong branching at all nodes up to a specified depth d of the branch-and-bound tree (LINDO Systems Inc. [26]). The work of Linderoth and Savelsbergh is improved by Achterberg et al., in a process called *reliability branching* in which the choice of the set $\mathcal{C}$ and the number of pivots to perform is dynamically altered during the course of the algorithm (Achterberg et al. [3]).

In this paper, we consider the exact *opposite* question as that of previous authors. That is, instead of performing *less* work than full strong branching, what if we performed *more*? Specifically, by accounting for the impact of the current branching decision on the bounds of the descendent nodes *two* levels deeper than the current node, can we make better branching decisions? The intuition behind studying this question comes from viewing strong branching as a greedy heuristic for selecting the branching variable. By considering the impact of the branching decision not just on the child subproblems, but on the grandchild subproblems as well, can we do better? And if so, at what computational cost? Karzan et al. [22] have also recently introduced a branching rule that considers the combined impact of multiple branching decisions. Their work uses information about fathomed nodes from an incomplete search. As such, it falls into the family of branching methods that leverage logical information when making branching decisions (Achterberg [1], Achterberg and Berthold [2], Sandholm and Shields [32]).

Obviously, obtaining information about the bounds of potential child nodes two levels deeper than the current node may be computationally expensive. In this work, we will for the most part focus on the question of *if* attempting to obtain this information is worthwhile. A secondary consideration will be on how to obtain the information in a computationally efficient manner. However, even if obtaining this information is extremely costly, we note two factors that may mitigate this expense. First, in codes for mixed integer programming

that are designed to exploit significant parallelism by evaluating nodes of the branch-and-bound tree on distributed processors (Eckstein et al. [15], Linderoth [24], Ralphs [31]), in the initial stages of the algorithm, there are not enough active nodes to occupy available processors. If obtaining information about the impact of branching decisions at deeper levels of the tree is useful, then these idle processors could be put to useful work by computing this information. Second, as noted by numerous authors (Forrest et al. [17], Linderoth and Savelsbergh [25]), the branching decisions made at the top of the tree are the most crucial. Perhaps the "expensive" lookahead branching techniques need only be done at for the very few first nodes of the branch-and-bound tree.

We are not aware of a work that considers the impact of the branching decision on grandchild nodes. Anstreicher and Brixius consider a "weak" (but computationally efficient) method of two-level branching as one of four branching methods described in Brixius and Anstreicher [9]. In the method, k_1 "pivots" are made to consider one branching decision; then, using dual information akin to the penalties of Driebeek [13], one more "pivot" on a second branching entity is considered. The current paper is an abbreviated version of the Master's Thesis of Glankwamdee [20], wherein more complete computational results can be found.

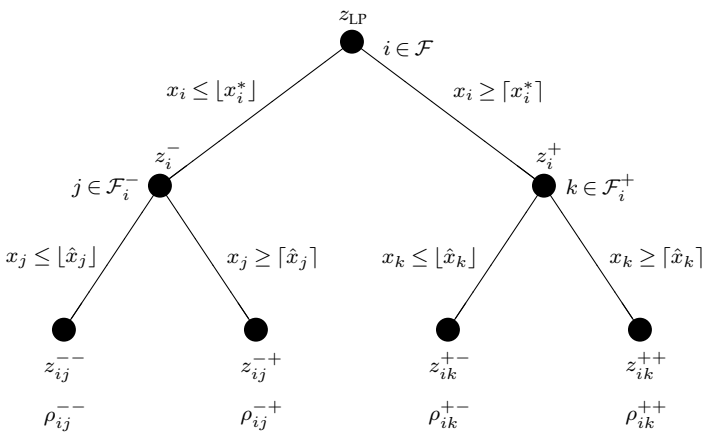
The paper has three remaining sections. In §2, we explain the method for gathering branching information from child and grandchild nodes, and we give one way to use this information to make a branching decision. We also show that auxiliary information from the branching variable determination process can be used to tighten the LP relaxation and reduce the size of the search tree. Section 3 presents methods to speed up the lookahead branching method. Extensive computational experiments are performed to determine good parameter settings for practical strong branching and lookahead methods. These branching methods are compared to that of MINTO, a sophisticated solver for mixed integer programs. Conclusions are offered in §4.

2. Lookahead Branching

In this section, we examine the question of whether or not significantly useful branching information can be obtained from potential grandchild nodes in the branch-and-bound tree. We explain our method for gathering this information and describe a simple lookahead branching rule that hopes to exploit the branching information obtained.

Figure 1 shows a potential two-level expansion of the search tree from an initial node. The set $\mathcal{F}$ is the set of fractional variables in the solution to the initial LP relaxation (x^*). By definition, the solution value of an infeasible linear program is denoted as $z_{\text{LP}} = -\infty$, and

FIGURE 1. Notations for lookahead search tree.



the best lower bound on the optimal solution value z_{MIP} is denoted at z_L. If the constraint $x_i \leq \lfloor x_i^* \rfloor$ is imposed on the left branch, and the relaxation is resolved, a solution of value z_i^- is obtained, and there is a set of variables $\mathcal{F}_i^- \subseteq I$ that takes fractional values. We use the parameter $\xi_i^- = 1$ to indicate if the left branch would be pruned (i.e., if $z_i^- \leq z_L$); otherwise $\xi_i^- = 0$. Similarly, if the bound constraint $x_i \geq \lceil x_i^* \rceil$ is imposed on the right branch, a solution of value z_i^+ is obtained, a set of variables $\mathcal{F}_i^+ \subseteq I$ is fractional, and the parameter ξ_i^+ indicates if the child node would be pruned.

Continuing to the second level in Figure 1, if the variable $j \in \mathcal{F}_i^-$ is chosen as the branching variable for the left child node, then the solution values for the two grandchild nodes are denoted as z_{ij}^{--} and z_{ij}^{-+}, and the indicator parameters ρ_{ij}^{--} and ρ_{ij}^{-+} are set to 1 if the corresponding grandchild nodes would be pruned, otherwise the indicators are set to 0. The notation for grandchild nodes on the right is similar.

2.1. Branching Rules

Once the branching information from the grandchild nodes is collected, there is still the question of to how to use this information to aid the current branching decision. Two reasonable objectives in choosing a branching variable are to minimize the number of grandchild nodes that are created and to try to decrease the LP relaxation bounds at the grandchild nodes as much as possible. Various combinations of these objectives are explored in Glankwamdee [20]. To keep the exposition short, we mention only one such method here. We use the following definitions. Let

$$\mathcal{G}_i^- \stackrel{\text{def}}{=} \{j \in \mathcal{F}_i^- \mid \rho_{ij}^{--} = 0,\, \rho_{ij}^{-+} = 0\} \quad \text{and} \tag{1}$$

$$\mathcal{G}_i^+ \stackrel{\text{def}}{=} \{k \in \mathcal{F}_i^+ \mid \rho_{ik}^{+-} = 0,\, \rho_{ik}^{++} = 0\} \tag{2}$$

be the sets of indices of fractional variables in child nodes both of whose grandchild nodes would not be pruned. To combine the progress on bound reduction of two child nodes into one number, we use the weighting function

$$\mathcal{W}(a,b) \stackrel{\text{def}}{=} \{\mu_1 \min(a,b) + \mu_2 \max(a,b)\}, \tag{3}$$

as suggested by Eckstein [14]. In this paper, the parameters of the weighting function are set to $\mu_1 = 4$ and $\mu_2 = 1$. Linderoth and Savelsbergh verifies empirically that these weights resulted in good behavior over a wide range of instances (Linderoth and Savelsbergh [25]). Let the reduction in the LP relaxation value at the grandchild nodes be denoted by

$$D_{ij}^{s_1 s_2} \stackrel{\text{def}}{=} z_{\text{LP}} - z_{ij}^{s_1 s_2}, \quad \text{where } s_1, s_2 \in \{-,+\}. \tag{4}$$

Note that $D_{ij}^{s_1 s_2} \geq 0$. The symbol η_i counts the total number of potential grandchild nodes that would be fathomed if variable i is chosen as the branching variable i.e.,

$$\eta_i \stackrel{\text{def}}{=} \sum_{j \in \mathcal{F}_i^-} (\rho_{ij}^{--} + \rho_{ij}^{-+}) + \sum_{k \in \mathcal{F}_i^+} (\rho_{ik}^{+-} + \rho_{ik}^{++}). \tag{5}$$

The two goals of branching, bound reduction and node elimination, are combined into one measure through a weighted combination. The branching rule employed in the experiments chooses to branch on the variable i^* that maximizes this weighted combination, namely

$$i^* \in \operatorname*{argmax}_{i \in \mathcal{F}} \left\{ \max_{j \in \mathcal{F}_i^-} \{\mathcal{W}(D_{ij}^{--}, D_{ij}^{-+})\} + \max_{k \in \mathcal{F}_i^+} \{\mathcal{W}(D_{ik}^{+-}, D_{ik}^{++})\} + \lambda \eta_i \right\}, \tag{6}$$

$$\text{where} \quad \lambda = \frac{1}{|\mathcal{G}_i^-|} \sum_{j \in \mathcal{G}_i^-} \mathcal{W}(D_{ij}^{--}, D_{ij}^{-+}) + \frac{1}{|\mathcal{G}_i^+|} \sum_{k \in \mathcal{G}_i^+} \mathcal{W}(D_{ik}^{+-}, D_{ik}^{++}) \tag{7}$$

is the average (weighted) reduction in LP value of all potential grandchild nodes in the sets $\mathcal{G}_i^-$ and $\mathcal{G}_i^+$. This value of λ is chosen to give the terms in Equation (6) the same scale. Note that in Equation (6), the variables $j \in \mathcal{F}_i^-$ and $k \in \mathcal{F}_i^+$ that maximize the weighted degradation of the grandchild nodes LP relaxation value may be different. To implement full strong branching, we let

$$D_i^s \stackrel{\text{def}}{=} z_{\text{LP}} - z_i^s, \quad \text{where } s \in -, +, \tag{8}$$

and we branch on the variable

$$i^* \in \operatorname*{argmax}_{i \in \mathcal{F}} \{\mathcal{W}(D_i^-, D_i^+)\}. \tag{9}$$

2.2. Implications and Bound Fixing

When computing the LP relaxation values for many potential child and grandchild nodes, auxiliary information that can be useful for tightening the LP relaxation and reducing the size of the search tree is obtained.

2.2.1. Bound Fixing. When tentatively branching on a variable x_i, either in strong branching or in lookahead branching, if one of the child nodes is fathomed, then the bounds on variable x_i can be improved. For example, if the child node with branching constraint $x_i \geq \lceil x_i^* \rceil$ is infeasible ($\xi_i^+ = 1$), then we can improve the upper bound on variable i to be $x_i \leq \lfloor x_i^* \rfloor$. Likewise, if there exists no feasible integer resolution for a variable j after branching on a variable i, then the bound on variable i can be set to its complementary value. The exact conditions under which variables can be fixed are shown in Table 1.

2.2.2. Implications. By examining consequences of fixing 0-1 variables to create potential grandchild nodes, simple inequalities can be deduced by combining mutually exclusive variable bounds into a single constraint. The inequality identifies two variables, either original or complemented, that cannot simultaneously be 1 in an optimal solution. For example, if variables x_i and x_k are binary decision variables, and the lookahead branching procedure determines that branching "up" on both x_i and x_k (i.e., $x_i \geq 1$, $x_k \geq 1$) results in a subproblem that may be pruned, then the inequality $x_i + x_k \leq 1$ can be safely added to the LP relaxation at the current node and all descendant nodes. These inequalities are essentially additional edges in a local *conflict graph* for the integer program (Atamtürk et al. [5], Savelsbergh [33]). As a line of future research, we intend to investigate the impact of adding these edges to the local conflict graph and performing additional preprocessing. Further *grandchild inequalities* can be added if any of the grandchild nodes are pruned as specified in Table 2.

2.3. Experimental Setup

The lookahead branching rule has been incorporated into the mixed integer optimizer MINTO v3.1, using the `appl_divide()` user application function that allows the user to specify the branching variable (Nemhauser et al. [30]). In all the experiments, the default

Table 1. Bound fixing conditions.

Condition	Implication
$\xi_i^- = 1$	$x_i \geq \lceil x_i^* \rceil$
$\xi_i^+ = 1$	$x_i \leq \lfloor x_i^* \rfloor$
$\rho_{ij}^{--} = 1$ and $\rho_{ij}^{-+} = 1$	$x_i \geq \lceil x_i^* \rceil$
$\rho_{ik}^{+-} = 1$ and $\rho_{ik}^{++} = 1$	$x_i \leq \lfloor x_i^* \rfloor$

Table 2. Grandchild implications.

Condition	Inequality
$\rho_{ij}^{--} = 1$	$(1 - x_i) + (1 - x_j) \leq 1$
$\rho_{ij}^{-+} = 1$	$(1 - x_i) + x_j \leq 1$
$\rho_{ik}^{+-} = 1$	$x_i + (1 - x_k) \leq 1$
$\rho_{ik}^{++} = 1$	$x_i + x_k \leq 1$

MINTO option, including preprocessing and probing, automatic cut generation, and reduced cost fixing, are used. The focus of the lookahead branching method is to reduce the upper bounds of the relaxations obtained after branching. We will measure the quality of a branching method by the number of nodes in the branch-and-bound tree. Therefore, for these experiments, the lower bound value z_L is initialized to be objective value of the (known) optimal solution. By setting z_L to the value of the optimal solution, we minimize factors other than branching that determine the size of branch-and-bound tree. To solve the linear programs that arise, we use CPLEX (v9.1) (CPLEX Optimization [10]). To speedup the testing of the algorithm, we run the experiments on a Beowulf cluster at Lehigh University. The code is compiled with gcc version 2.96 (Red Hat Linux 7.1) and run on Intel(R) Pentium(R) III CPU, with clock speed 1133 MHz. The CPU time is limited to a maximum of 8 hours, and the memory is limited to a maximum of 1024 MB. We have limited initial test to a suite of 16 instances from MIPLIB 3.0 (Bixby et al. [7]) and MIPLIB 2003 (Martin et al. [27]).

2.4. Computational Results

Our first experiment runs an implementation of full strong branching, with and without bound fixing and implications, and lookahead branching, with and without bound fixing and implications. The focus of the experiment is not on the speed of the resulting methods, but instead on the following two questions:

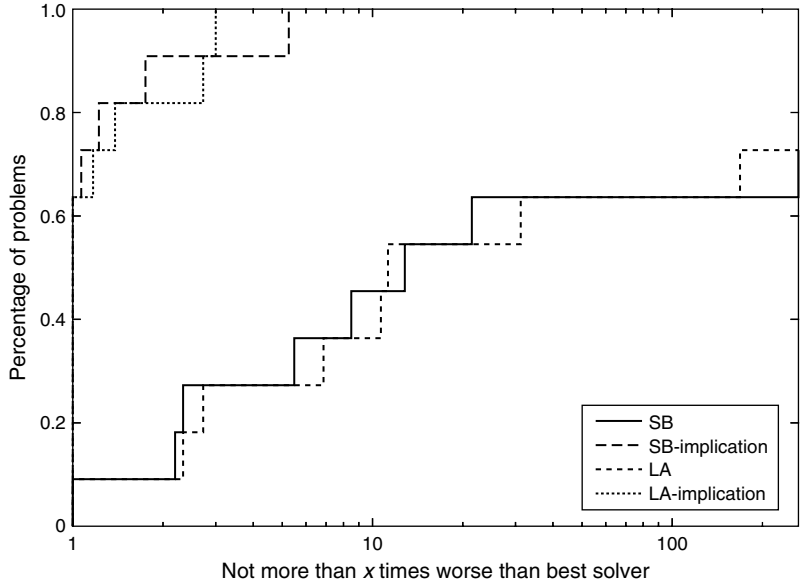
- Does lookahead branching often make different branching decisions compared to full strong branching? If so, what are the positive impacts of these branching decisions?
- Do bound fixing and grandchild inequalities coming from implications found in the lookahead branching procedure have a positive impact on the size of the search tree?

Tables A.1, A.2, and A.3 in the appendix display full details of the experimental runs. To summarize the results of the experiments, we use performance profiles plotted in log scale, as introduced by Dolan and Moré [12]. A performance profile is a relative measure of the effectiveness of a solver s when compared to a group of solvers $\mathcal{S}$ on a set of problem instances P. To completely specify the performance profile, we need the following definitions:

- γ_{ps} is a quality measure of solver s when solving problem p,
- $r_{ps} = \gamma_{ps}/(\min_{s \in \mathcal{S}} \gamma_{ps})$, and
- $\rho_s(\tau) = |\{p \in P \mid r_{ps} \leq \tau\}|/|P|$.

Hence, $\rho_s(\tau)$ is the fraction of instances for which the performance of solver s is within a factor of τ of the best. A performance profile for solver s is the graph of $\rho_s(\tau)$. In general, the higher the graph of a solver, the better the relative performance. Eleven of the sixteen instances are solved to provable optimality by one of the four methods, and for these instances, we use the number of nodes as the quality measure γ_{ps}. Under this measure, $\rho_s(1)$ is the fraction of instances for which solver s evaluated the fewest number of nodes to verify optimality, and $\rho_s(\infty)$ is the fraction of instances for which solver s verified the optimality of the solution of value z_L. Figure 2 shows the performance profile plot for these eleven instances. *SB* and *LA* denote strong branching and lookahead branching respectively while *Implication* indicates that bound fixing and implications are added to the algorithms. Two conclusions can evidently be drawn from Figure 2.

FIGURE 2. Performance profile of number of evaluated nodes in solved instances.



(1) Using bound fixing and grandchild inequalities can *greatly* reduce the number of nodes in the branch-and-bound tree.

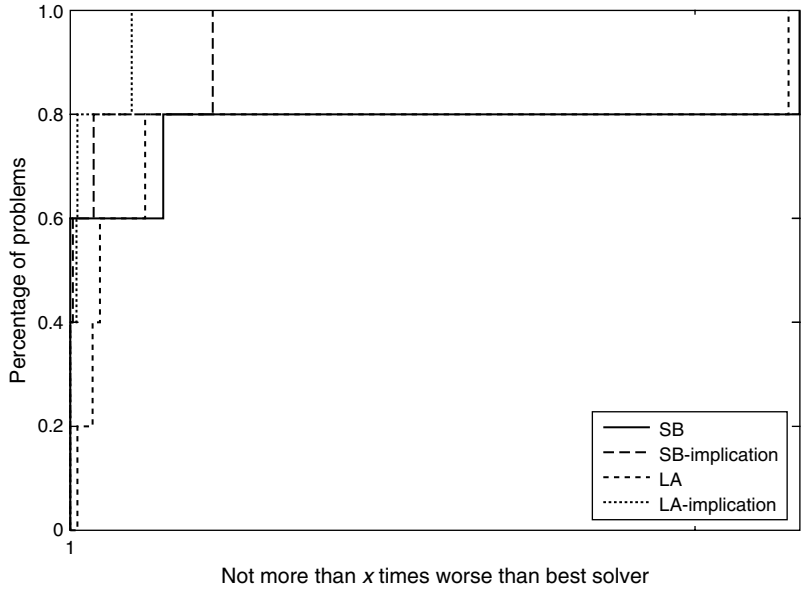
(2) Neither full strong branching nor lookahead branching seems to significantly outperform the other in these tests.

The fact that full strong branching and lookahead branching seem to be of comparable quality is slightly surprising, more so when one considers the fact that lookahead branching *quite often* chooses to branch on a different variable than full strong branching does. In Table 3, the second column lists the percentage of nodes at which lookahead branching and full strong branching would make *different* branching decisions. For example, for the instance `danoint`, the two methods choose a different branching variable 86% of the time.

TABLE 3. Lookahead branching statistics.

Name	Percentage of difference	No. of bound fixing (per node)	No. of inequalities (per node)
aflow30a	76	1.53	47.80
aflow40b	59	0.72	26.24
danoint	86	0.85	3.48
l152lav	25	2.56	267.56
misc07	73	1.32	32.35
modglob	50	1.00	17.13
opt1217	46	1.00	0.00
p0548	100	3.00	15.33
p2756	0	0.67	15.00
pk1	53	0.87	10.71
pp08a	75	1.02	0.86
qiu	100	3.60	120.60
rgn	71	0.64	2.75
stein45	60	1.19	72.08
swath	84	0.73	1.51
vpm2	54	0.83	8.81

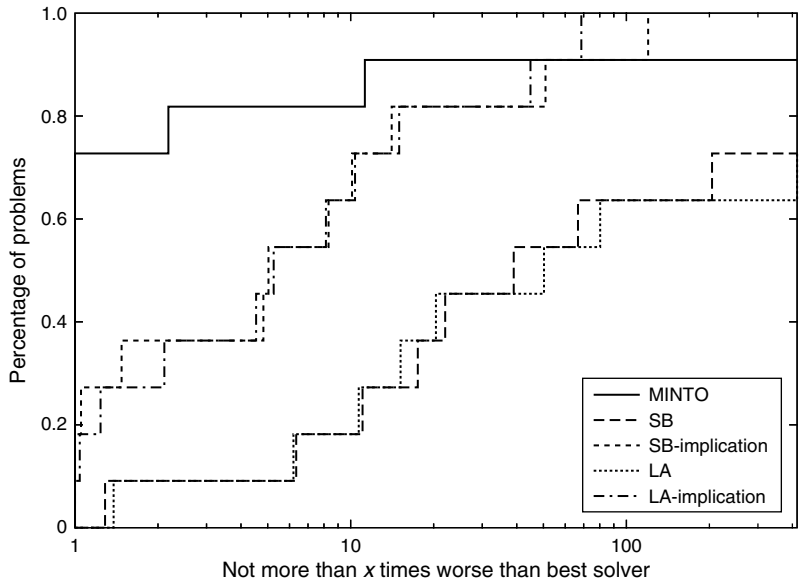
FIGURE 3. Performance profile of integrality gap in unsolved instances.



Also in Table 3, we list the number of times a variable's bound is improved per node, and the number of grandchild inequalities per node that is added.

For five of the sixteen instances, none of the branching methods is able to prove the optimality of the solution. For these instances, we use the quality measure (γ_{ps}), the final integrality gap after eight hours of CPU time. Figure 3 shows the performance profile of the four branching methods on the unsolved instances. Again, we see that the bound fixing and grandchild (implications) inequalities can be very effective, and that lookahead branching and full strong branching are of similar quality for these instances.

FIGURE 4. Performance profile of running time in solved instances.



The final performance profile (in Figure 4) uses the solution time as the quality parameter for the eleven solved instances and also includes a profile for the default MINTO branching rule. The profile demonstrates:

- As expected, the running time of the strong branching and lookahead branching are in general worse than the default MINTO.
- However, the added implications and bound fixing help to solve the `pk1` instance which is unsolved with the default MINTO.

3. Abbreviated Lookahead Branching

The initial experiment led us to believe that measuring the impact on grandchild nodes when making a branching decision can reduce the number of nodes of the search tree, in large part due to additional bound fixing and implications that can be derived. However, the time required to perform such a method can be quite significant. Our goal in this section is to develop a practical lookahead branching method. An obvious way in which to speed the process up is to consider fixing bounds on only certain pairs of variables and then to limit the number of simplex iterations used to gauge the change in bound at the resulting grandchild nodes.

3.1. Algorithm

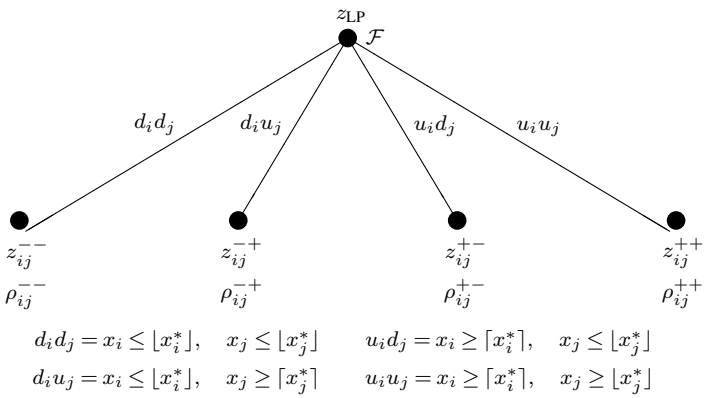
To describe the method employed, we use similar notation as for the original method described in §2. Figure 5 shows the notation we use for the values of the LP relaxations of the partially expanded search tree and the indicator variables if a particular grandchild node would be fathomed.

For a pair of variables (x_i, x_j) whose values in the current LP relaxation are fractional, we create the four subproblems denoted in Figure 5 and do a limited number of dual simplex pivots in order to get an upper bound on the values z_{ij}^{--}, z_{ij}^{-+}, z_{ik}^{+-}, and z_{ik}^{++}. The obvious questions we must answer are *how* to choose the pair of variables (x_i, x_j), and *how many* simplex pivots should be performed.

3.2. Strong Branching Implementation

The questions of how to choose candidate variables and how many pivots to perform on each candidate must also be answered when implementing a (one-level) strong branching method. Since our goal is to show possible benefits of the lookahead method, we also implement a practical strong branching method with which to compare the abbreviated

FIGURE 5. Notations for abbreviated lookahead search tree.



lookahead method. When implementing strong branching, there are two main parameters of interest:

- the number of candidate variables to consider, (or the size of the set $\mathcal{C}$), and
- the number of simplex pivots (down and up) to perform on each candidate.

Our choice to limit the size of the candidate set is based on how many fractional variables there are to consider in a solution $(\hat{x}, \hat{y})$ whose objective value is z_{LP}. Specifically, we let the size of this set be

$$|\mathcal{C}| = \max\{\alpha|\mathcal{F}|, 10\}, \tag{10}$$

where $\mathcal{F}$ is the set of fractional variables in $\hat{x}$, and $0 \leq \alpha \leq 1$ is a parameter whose value we will determine through a set of experiments. When branching, the variables are ranked from largest to smallest according to the fractionality of $\hat{x}_i$, i.e., the criteria $\min(f_i, 1 - f_i)$, where $f_i = \hat{x}_i - \lfloor \hat{x}_i \rfloor$ is the fractional part of $\hat{x}_i$. The top $|\mathcal{C}|$ variables are chosen as potential branching candidates. For each candidate variable x_i, β dual simplex iterations are performed for each of the down and up branch, resulting in objective values z_i^- and z_i^+. The variable selected for branching is the one with

$$i^* \in \underset{i \in \mathcal{F}}{\text{argmax}}\{\mathcal{W}(z_{\text{LP}} - z_i^-, z_{\text{LP}} - z_i^+)\}. \tag{11}$$

More sophisticated methods exist for choosing the candidate set $\mathcal{C}$. For example, the variables could be ranked based on the bound change resulting from one dual simplex pivot (akin to the penalty method of Driebeek [13]), or even a dynamic method, in which the size of the set considered is a function of the bound changes seen on child nodes to date, like the method of Achterberg et al. [3]. We denote by $\text{SB}(\hat{\alpha}, \hat{\beta})$ the strong branching method with parameters $\hat{\alpha}$ and $\hat{\beta}$. In our experiments, strong branching is implemented using the CPLEX routine `CPXstrongbranch()` (CPLEX Optimization [10]).

3.3. Lookahead Branching Implementation

When implementing the abbreviated lookahead branching method, we must determine

- the number of candidate variable pairs to consider, and
- the number of simplex iterations to perform on each of the four grandchild nodes for each candidate pair.

Our method for choosing the set of candidate pairs $\mathcal{D}$ works as follows. First, a limited strong branching $\text{SB}(\hat{\alpha}, \hat{\beta})$ is performed, as described in §3.2. Then, the variables are ranked from largest to smallest using the same criteria as in strong branching, namely $\mathcal{W}(z_{\text{LP}} - z_i^-, z_{\text{LP}} - z_i^+)$. From these, the best γ candidates are chosen, and for *each* pair of candidate variables coming from the best γ, δ dual simplex iterations are performed on the four grandchild nodes, resulting in the values $z_{ij}^{s_1 s_2}$ and $\rho_{ij}^{s_1 s_2}$ of Figure 5. If $\hat{\alpha}$ and $\hat{\beta}$ are the parameters for the limited strong branching, and $\hat{\gamma}$, $\hat{\delta}$ are the parameters defining the size of the candidate set of variable pairs and number of pivots on each grandchild node to perform, then we will refer to the branching method as $\text{LA}(\hat{\alpha}, \hat{\beta}, \hat{\gamma}, \hat{\delta})$. Note that the set $\mathcal{D}$ consists of all pairs of the best γ candidate variables from the limited strong branching. It may not be necessary to consider each pair, and we will consider other mechanisms for choosing the variable pairs as a line of future research.

3.4. Computational Results

A first set of experiments is performed to determine good values for the branching parameters α, β, γ, and δ. Subsequently, we compare the resulting strong branching and abbreviated lookahead branching methods with the default branching scheme of MINTO v3.1. MINTO v3.1 uses a combination of (once-initialized) pseudocosts and the penalty method of Driebeek [13]. See Linderoth and Savelsbergh [25] for a complete explanation of this method. For these experiments, we use a test suite of 89 instances from MIPLIB 3.0 (Bixby et al. [7]) and MIPLIB 2003 (Martin [27]). Besides the test suite of problems, all other characteristics of these experiments are the same as those described in §2.3.

3.4.1. Strong Branching Parameters. Our first goal is to determine reasonable values for α and β to use in our strong branching method SB(α, β). Doing a search of the full parameter space for α and β would have required prohibitive computational effort, so instead we employ the following mechanism for determining reasonable default values for α and β. The number of simplex iterations is fixed to $\beta = 5$, and an experiment is run to determine a good value of α given that $\beta = 5$. An experiment was run for values $\alpha = 0.25, 0.5, 0.75$, and 1.0. Details of this experiment, including a performance profile plot, can be found in the technical report version of this work (Glankwamdee and Linderoth [21]). The result of the experiment shows that $\alpha = 0.5$, considering half of the fractional variables as branching candidates, gives good results.

Next, we run an experiment comparing the branching rules SB$(0.5, \beta)$ for $\beta = 5, 10$, and 25. Again, details of this experiment can be found in the technical report (Glankwamdee and Linderoth [21]). The results of the experiment are inconclusive in determining a best value for the parameter β, but the value $\beta = 10$ appears to perform reasonably well.

3.4.2. Lookahead Branching Parameters. This experiment is designed to determine appropriate values for the number of branching candidates and the number of simplex pivots, i.e., parameters γ and δ respectively, in the abbreviated lookahead branching method, LA$(\alpha, \beta, \gamma, \delta)$. In this experiment, we have fixed the values for $(\alpha^*, \beta^*) = (0.5, 10)$ as determined in §3.4.1. To find appropriate values for γ and δ, we follow the similar strategy to the one that is used to determine α^* and β^*. First, we fix a value of $\delta = 10$ and compare the performance of branching rules LA(0.5, 10, γ, 10). The results of this experiment are summarized in Figure 6. We conclude from the experiment that $\gamma = 3$ is a reasonable parameter value.

Given that $\gamma = 3$, the next experiment compared branching methods LA$(0.5, 10, 3, \delta)$ for $\delta \in \{5, 10, 15, 20, 25\}$. The results of this experiment are summarized in the performance profile in Figure 7. There is also no clear winner in this experiment, but we prefer the smaller number of simplex pivots. Therefore, we select the good parameter settings for the abbreviated lookahead branching method to be $(\alpha^*, \beta^*, \gamma^*, \delta^*) = (0.5, 10, 3, 10)$.

3.4.3. Full Strong Branching and Abbreviated Lookahead Branching Comparison. This experiment is aimed at determining if the limited grandchild information obtained from the abbreviated lookahead branching method could reduce the number of

FIGURE 6. Performance profile of running time as γ varies.

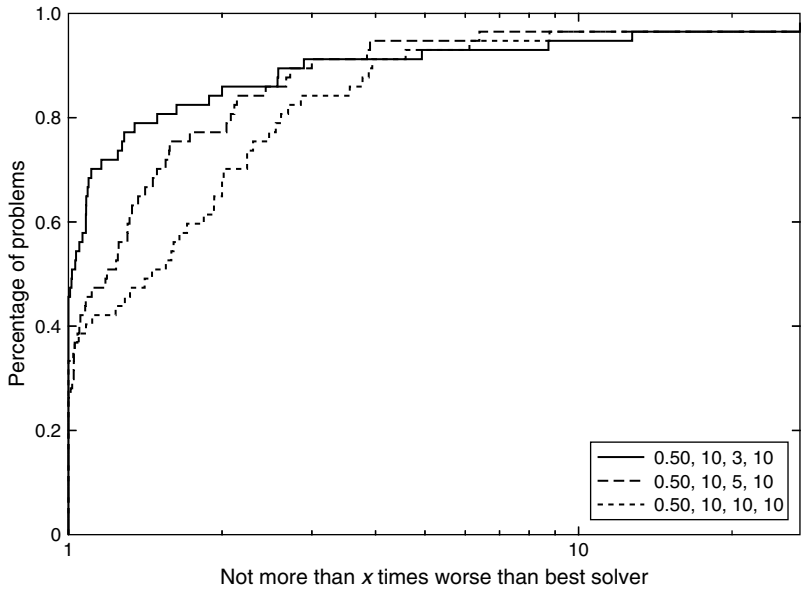
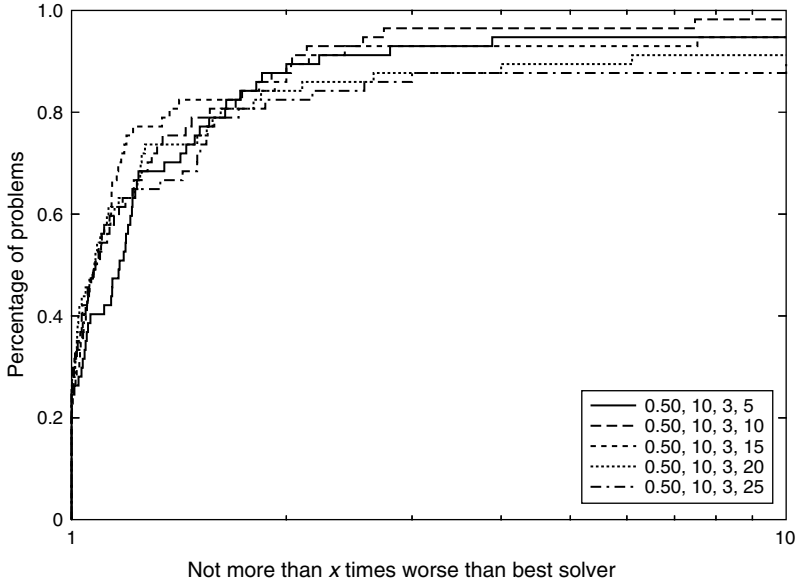


FIGURE 7. Performance profile of running time as δ varies.



nodes in the search tree significantly. Namely, we compare the branching methods SB(α^*, β^*) and LA$(\alpha^*, \beta^*, \gamma^*, \delta^*)$. The abbreviated lookahead branching method will not be at all effective if the number of nodes in LA$(\alpha^*, \beta^*, \gamma^*, \delta^*)$ is not significantly less than SB(α^*, β^*) since the amount of work done to determine a branching variable is significantly larger in LA$(\alpha^*, \beta^*, \gamma^*, \delta^*)$ than for SB(α^*, β^*).

Figure 8 is the performance profile comparing the number of nodes evaluated in the two methods. The number of evaluated nodes in the abbreviated lookahead method is substantially less than the strong branching. A somewhat more surprising result is depicted in Figure 9, which shows that LA$(\alpha^*, \beta^*, \gamma^*, \delta^*)$ also dominates SB(α^*, β^*) in terms of CPU time.

FIGURE 8. Performance profile of number of evaluated nodes.

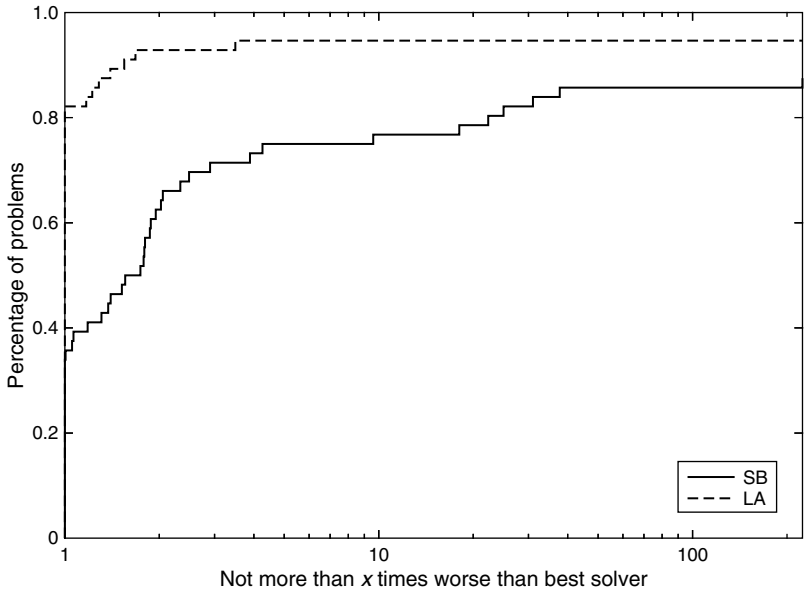
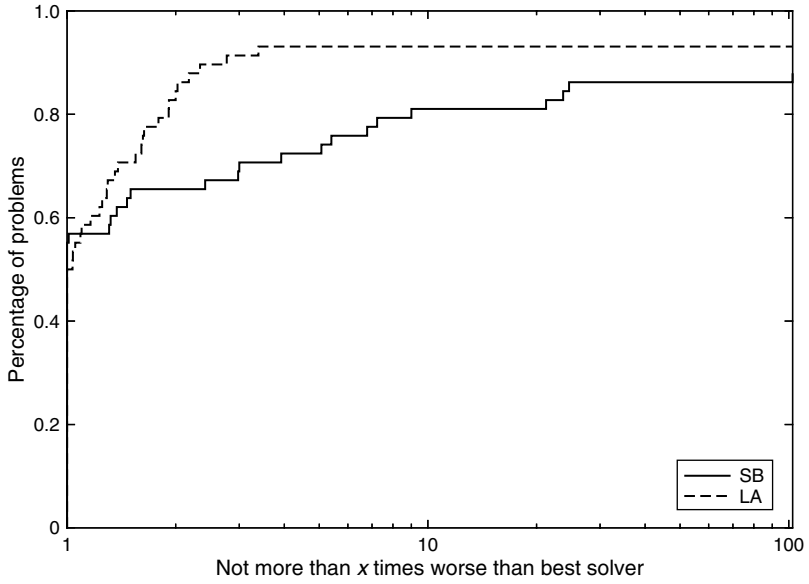


FIGURE 9. Performance profile of running time.



3.4.4. Final Comparison. The final experiment is the most practical one, aimed at determining for a fixed maximum number of simplex iterations, whether these iterations are most effectively used evaluating potential child node bounds or grandchild node bounds. Namely, we would like to compare the strong branching strategy against the abbreviated lookahead branching strategy for parameter values such that

$$2|\mathcal{C}_1|\beta = 4\frac{\gamma(\gamma-1)\delta}{2}. \tag{12}$$

The LHS of Equation (12) is the maximum number of pivots that the strong branching method will perform, where $|\mathcal{C}_1|$ is computed from Equation (10). Similarly, the RHS is the maximum number of pivots that the lookahead branching method can perform.

For this experiment, we implement the strong branching with bound fixing and the abbreviated lookahead branching with bound fixing as part of MINTO options. The variables in the set $\mathcal{C}$ are ranked as described earlier according to the fractionality while the variables in the set $\mathcal{D}$ are ranked from largest to smallest using the pseudocosts. We use CLP (Forrest [16]) to solve the linear programs. The strong branching parameters are fixed at the setting determined in §3.4.1, namely $\alpha^* = 0.5$ and $\beta^* = 10$. We fix the lookahead branching parameter $\delta^* = 10$ as determined in §3.4.2. Finally, γ is computed from Equation (12).

We compare the effectiveness of these methods with the default MINTO branching. We also implement another branching strategy, LA-5, that applies the abbreviated lookahead branching at the top of branch-and-bound tree, i.e., only to nodes with depth less than 5, and use the default MINTO scheme otherwise. The experimental results are summarized in the performance profiles of Figures 10 and 11. Full details of the experimental runs can be found in Table A.4 in the appendix. For a fixed number of simplex iterations, the abbreviated lookahead branching outperforms the strong branching. In addition, the abbreviated lookahead method leads to fewer evaluated nodes.

The average CPU time for MINTO on the instances in the test suite was 730.21 seconds, while the average for the lookahead branching method was 558.81 seconds. The geometric mean of the CPU time for MINTO was 9.55 seconds, and the geometric mean for the lookahead branching was larger, 10.17 seconds. The fact that the average time for lookahead branching is smaller, while the geometric mean is larger is an indication that performing lookahead branching may prove beneficial on difficult instances.

FIGURE 10. Performance profile of number of evaluated nodes.

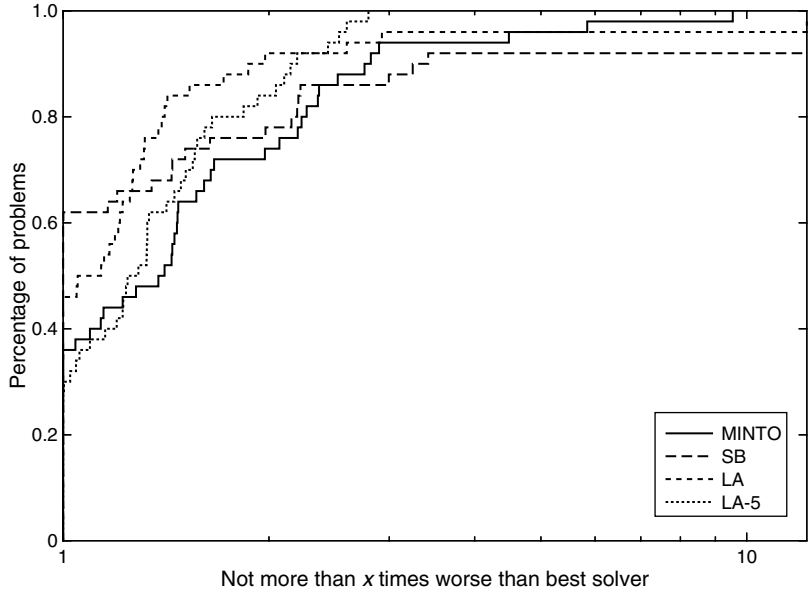
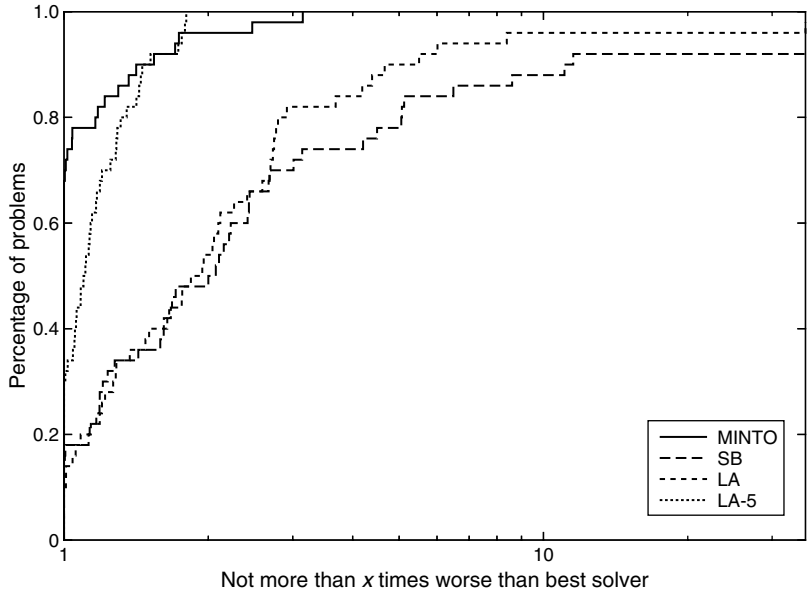


FIGURE 11. Performance profile of running time.



4. Conclusion

We have asked and partially answered the question of whether or not consider branching information from grandchild nodes can result in smaller search trees. We have proposed a method for gathering the information from grandchild nodes. We verified that this information can often be quite useful in reducing the total number of nodes in the search, can result in fixing bounds on variables, and can often give implications between variables. Finally, we show that by the limiting number of simplex iterations or the number of fractional variables for which to generate the branching information, a similar branching decision can still be made, but with less computational effort. The resulting branching rule is of comparable

quality to the advanced branching methods available in the MINTO software system. From our experience, it seems unlikely that lookahead branching will be a good default branching scheme for general MIP, but for some classes of hard problem instances, the additional computational effort may well pay off.

Appendix. Tables of Results

Table A.1. Solved MIPLIB instances.

	Number of evaluated nodes			
Name	SB	SB-implication	LA	LA-implication
l152lav	193	11	281	9
p0548	3	3	3	3
rgn	1,011	127	1,296	119
stein45	16,437	7,491	20,409	8,755
vpm2	4,883	381	4,291	527
misc07	8,173	163	5,219	31
modglob	159	29	199	79
p2756	7	3	7	3
aflow30a	−1	15	−1	45
pk1	−1	24,001	−1	13,731
qiu	−1	5	−1	5

Note. A value of −1 indicates that the instance is not solved within the time limit of eight hours.

Table A.2. Unsolved MIPLIB instances.

	Integrality gap			
Name	SB	SB-implication	LA	LA-implication
opt1217	23.88	23.88	24.07	24.07
pp08a	10.63	10.66	11.55	11.38
aflow40b	13.25	6.91	13.09	5.90
danoint	4.38	4.38	4.49	4.41
swath	32.47	30.66	30.27	29.29

Table A.3. Total running time of solved MIPLIB instances.

	Total running time				
Name	MINTO	SB	SB-implication	LA	LA-implication
l152lav	8.84	1,808.68	1,059.83	3,678.86	606.83
p0548	0.21	0.27	0.31	0.29	0.26
rgn	2.91	32.13	24.18	31.13	23.66
stein45	296.94	19,820.37	15,107.89	23,863.70	13,323.79
vpm2	23.71	522.02	144.39	483.05	107.62
misc07	790.96	13,852.87	8,000.36	11,993.02	8,201.39
modglob	2.95	115.14	41.55	148.21	44.25
p2756	2.92	18.52	14.70	18.10	15.34
aflow30a	1,842.31	−1	844.3	−1	1,779.79
pk1	−1	−1	4,928.79	−1	4,685.71
qiu	5,112.87	−1	454.33	−1	472.78

Note. A value of −1 indicates that the instance is not solved within the time limit of eight hours.

Table A.4. Number of evaluated nodes and CPU time in final comparison.

	Number of evaluated nodes				Total running time			
Name	MINTO	SB	LA	LA-5	MINTO	SB	LA	LA-5
aflow30a	23,293	16,955	11,241	14,483	1,524.07	2,119.58	1,320.62	877.76
air03	1	1	1	1	1.12	1.12	1.13	1.18
air04	1,093	187	321	301	1,457.67	747.19	1,047.84	462.89
air05	2,625	275	361	605	1,205.85	622.73	809	488.19
bell3a	44,779	44,811	47,029	44,911	30.16	30.28	34.3	31.72
bell5	8,243	−1	100,943	8,245	5.38	−1	189.42	5.83
blend2	1,527	1,085	2,143	1,613	8.27	22.21	38.58	10.68
cap6000	13,067	11,939	15,073	15,855	339.49	750.34	1,421.98	403.3
dcmulti	1,075	1,133	841	861	5.05	25.18	13.4	4.97
disctom	1	1	1	1	9.81	9.83	9.77	9.83
dsbmip	1	1	1	1	0.95	0.95	0.99	0.95
egout	3	3	3	3	0.02	0.02	0.02	0.02
enigma	1	1	1	1	0.01	0.01	0.01	0.01
fast0507	9,439	−1	−1	5,677	19,002.07	−1	−1	12,335.81
fiber	193	43	45	57	8.11	7.41	6.25	6.62
fixnet6	81	263	105	97	2.79	12.55	5.42	3.95
flugpl	5,295	7,103	3,593	5,933	1.32	5.55	3.62	1.5
gen	3	3	3	3	0.16	0.19	0.19	0.17
gesa2	79,739	−1	50,911	74,073	1,324.28	−1	2,595.98	1,272.3
gesa2_o	95,947	−1	42,243	81,313	1,125.75	−1	2,566.79	956.78
gesa3	1,249	2,327	777	821	32.99	208.02	67.57	24.18
gesa3_o	699	1,555	875	765	14.19	157.02	85.33	20.35
gt2	5	11	5	5	0.05	0.1	0.06	0.06
harp2	5,327	7,677	6,451	6,583	338.85	3,910.79	1,864.7	387.64
khb05250	13	13	13	13	0.39	0.47	0.5	0.5
l152lav	571	435	197	279	19.41	45.55	22.75	18.68
lseu	227	103	145	211	0.94	1.57	1.39	1.27
misc03	2,333	989	1,363	1,541	6.02	12.03	13.71	4.95
misc06	55	65	45	47	1.44	4.52	2.54	1.89
misc07	86,581	30,705	57,279	77,779	520.07	996.36	1,649.02	447.34
mitre	1	1	1	1	10.64	10.55	10.65	10.74
mod008	511	259	361	557	2.28	3.83	4.46	2.38
mod010	69	41	25	61	6.61	7.72	6.99	7.44
mod011	9,459	4,237	12,401	11,851	5,605.32	3,964.62	9,553.1	7,133.23
modglob	247	237	277	273	3.57	9.54	10.41	6.18
nw04	291	177	271	325	529.64	653.21	974.36	770.89
p0033	13	9	11	11	0.07	0.1	0.09	0.09
p0201	255	177	233	235	3.16	6.55	8.19	3.47
p0282	45	41	19	19	1.3	2.8	2.1	1.97
p0548	3	3	3	3	0.24	0.27	0.26	0.26
p2756	5	5	13	13	3.67	4.15	6.46	6.46
qiu	117,577	46,613	56,121	70,456	3,975.41	4,908.47	4,791.42	2,331.61
qnet1	95	325	135	147	5.13	25.92	13.99	9.17
qnet1_o	223	152	181	320	7.38	11.72	10.12	10.67
rentacar	51	43	37	49	19.82	32.55	25.1	22.03
rgn	2,657	2,319	2,635	2,881	7.02	12	14.73	8.19
stein27	3,939	2,707	3,107	3,615	2.4	12.29	10.54	2.81
stein45	56,861	38,681	48,937	60,763	84.66	549.29	709.91	95.69
vpm1	1	1	1	1	0.04	0.04	0.04	0.05
vpm2	9,307	9,839	8,201	10,041	49.48	148.98	104.83	54.82
Average	11,464.96	9,927.67	9,656.67	9,984.02	730.21	1,063.04	1,173.67	558.81
Geometric mean	298.93	260.29	250.57	267.95	9.55	18.93	17.73	10.17

Note. A value of −1 indicates that the instance is not solved within the time limit of eight hours.

References

[1] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization* 4:4–20, 2007.

[2] T. Achterberg and T. Berthold. Hybrid branching. W. J. van Hoeve and J. Hooker, eds. *CPAIOR: Lecture Notes in Computer Science*, Vol. 5547. Springer, Berlin, 309–311, 2009.

[3] T. Achterberg, T. Koch, and A. Martin. Branching rules revisited. *Operations Research Letters* 33:42–54, 2004.

[4] D. Applegate, R. Bixby, V. Chvátal, and W. Cook. On the solution of traveling salesman problems. *Documenta Mathematica Journal der Deutschen Mathematiker-Vereinigung, International Congress of Mathematicians* 3:645–656, 1998.

[5] A. Atamtürk, G. Nemhauser, and M. W. P. Savelsbergh. Conflict graphs in solving integer programming problems. *European Journal of Operational Research* 121:40–55, 2000.

[6] M. Bénichou, J. M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming* 1:76–94, 1971.

[7] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima* 58:12–15, 1998.

[8] R. Breu and C. A. Burdet. Branch and bound experiments in zero-one programming. *Mathematical Programming* 2:1–50, 1974.

[9] N. Brixius and K. Anstreicher. Solving quadratic assignment problems using convex quadratic programming relaxations. *Optimization Methods and Software* 16:49–68, 2001.

[10] CPLEX Optimization. Using the CPLEX Callable Library, Version 9.1. CPLEX Optimization, Inc., Incline Village, NV, 2005.

[11] Dash Optimization. XPRESS-MP Reference Manual. Dash Optimization, Ltd., Leamington, Spa, UK, 2006.

[12] E. Dolan and J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming* 91:201–213, 2002.

[13] N. J. Driebeek. An algorithm for the solution of mixed integer programming problems. *Management Science* 12:576–587, 1966.

[14] J. Eckstein. Parallel branch-and-bound methods for mixed integer programming. *SIAM News* 27:12–15, 1994.

[15] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO: An object-oriented framework for parallel branch-and-bound. *Proc. Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications*, 219–265, 2001.

[16] J. Forrest. CLP. http://www.coin-or.org/. 2004. Accessed June 1, 2004.

[17] J. J. H. Forrest, J. P. H. Hirst, and J. A. Tomlin. Practical solution of large scale mixed integer programming problems with UMPIRE. *Management Science* 20:736–773, 1974.

[18] J. M. Gauthier and G. Ribière. Experiments in mixed-integer linear programming using pseudocosts. *Mathematical Programming* 12:26–47, 1977.

[19] A. M. Geoffrion and R. E. Marsten. Integer programming algorithms: A framework and state-of-the-art survey. *Management Science* 18:465–491, 1972.

[20] W. Glankwamdee. Lookahead branching for mixed integer programming. Master's thesis, Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA, 2004.

[21] W. Glankwamdee and J. T. Linderoth. Lookahead branching for mixed integer programming. Technical Report 06T-004, Department of Industrial and Systems Engineering, Lehigh University, Bethlehem, PA, 2006.

[22] F. K. Karzan, G. L. Nemhauser, and M. W. P. Savelsbergh. Information-based branching schemes for binary linear mixed integer problems. *Mathematical Programming Computation* 1:249–293, 2009.

[23] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica* 28:497–520, 1960.

[24] J. T. Linderoth. Topics in Parallel Integer Optimization. Ph.D. thesis, Georgia Institute of Technology, Atlanta, 1998.

[25] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies in mixed integer programming. *INFORMS Journal on Computing* 11:173–187, 1999.

[26] LINDO Systems. LINDO User's Manual, LINDO Systems Inc., Chicago, 1998.

[27] A. Martin, T. Achterberg, and T. Koch. MIPLIB 2003. http://miplib.zib.de, 2003.

[28] G. Mitra. Investigation of some branch and bound strategies for the solution of mixed integer linear programs. *Mathematical Programming* 4:155–170, 1973.

[29] G. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. John Wiley and Sons, New York, 1988.

[30] G. L. Nemhauser, M. W. P. Savelsbergh, and G. C. Sigismondi. MINTO, a mixed INTeger optimizer. *Operations Research Letters* 15:47–58, 1994.

[31] T. K. Ralphs. SYMPHONY 5.0 User's Manual, November 11, 2004. http://projects.coin-or.org/svn/SYMPHONE/html/doc/SYMPHONY-5.0-Manual.pdf. Accessed December 2, 2010.

[32] T. Sandholm and R. Shields. Nogood learning for mixed integer programming. Technical Report CMU-CS-06-155, Computer Science Department, Carnegie Mellon University, Pittsburgh, 2006.

[33] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing* 6:445–454, 1994.

[34] L. A. Wolsey. *Integer Programming*. John Wiley and Sons, New York, 1998.